Formal Analysis of Safety Critical System

Research Proposal

Candidates name	:	**Monika Singh**
Enrollment Number	:	**110433**
Supervisor	:	**Prof. (Dr.) A. K. Sharma**

Mody Institute of Technology & Science
Faculty of Engineering & Technology
Lakshmangarh, Sikar – 332311 (Rajasthan)

Table of Content

Sr. No.	Topic	Page No.
1	Intoduction	1
1.1	Formal Method	1
1.2	Role of Formal Method in software development Process	1
1.3	Scope of Formal Methods	2
1.4	Formal Specification Languages	2
1.5	Type of Formal Specification Languages	4
1.6	Benefits of Formal Methods	5
1.7	Different levels of formal methods	5
2	Z-Notation	6
2.1	Syntax	6
2.2	Z Schemas	9
2.3	Petri Nets	11
3	Scope of Work	13
4	Objective and Proposed Approach	14
5	Time Plan	16
6	Conclusion	16

1. Intoduction

1.1Formal method

The advancements of the computing, control and communication technologies increased the complexity of hardware and software systems such as computing critical systems, automated systems, distributed system, networked or embedded systems etc. Moreover, the functionality and scale of such systems have also grown. Due to the increase in complexity, the likelihood of errors and flaws in the systems grown and design of ultimate bug-free systems becomes more difficult. Design of such a complex system can be possible with the help of advanced mathematical methodologies and step by step design process. Consequently, the issue of system's complexity attracts the attention of researchers towards the effective methods of modeling and verification of systems [2]. One of the best ways of achieving this goal is by using the formal methods. Formal methods are mathematical techniques, often supported by tools, for developing software and hardware systems. Mathematical rigour enables users to analyze and verify these models at any part of the program life-cycle: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution. The vital first step in a high-quality software development process is requirement engineering [1]. Formal methods can be useful in eliciting, articulating, and representing requirements. Their tools can provide automated support needed for checking completeness, traceability, verifiability, and reusability, and for supporting requirements evolution, diverse viewpoints, and inconsistency management [6].

1.2 Role of Formal method in Software Development process:

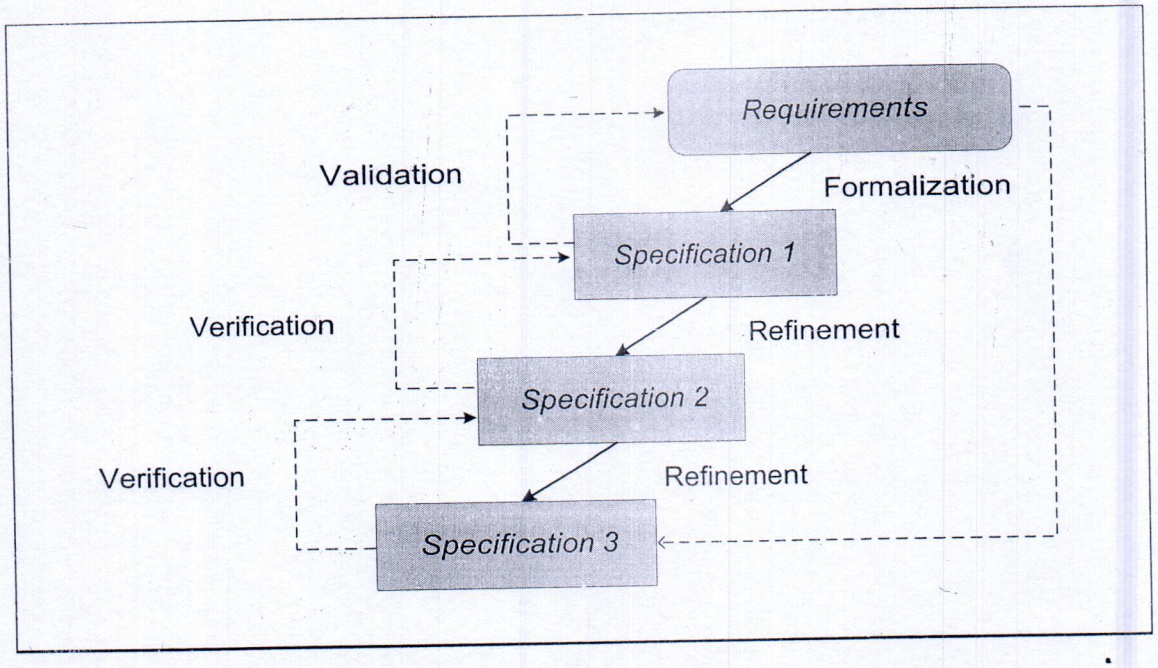

Fig1: Software Development Process using Formal Methods

Formal methods are languages which are depended on the use of mathematical procedures and notations to describe and analyze the properties of software systems [5]. These descriptions and notations are usually from the area of discrete mathematics, including set theory, predicate logic, and graph theory. These mathematical notations frees the software system from inconsistencies, ambiguities, and incompleteness. The procedure [4] of software development based on formal methods is described in Figure 1. Software life cycle model based on `refinement' is the result of a requirement analysis which is usually expressed in everyday language. `Specification 1' describes the formal requirements which are transformed from the informal language. Further, the development from `specification '1' to `specification n' describes the process of design. The two fundamental principles that arise in the development of a software system are the validation and verification. Validation describes whether the generated system accomplish the requirements, whereas verification ensures whether the created software meets the requirements found in the previous stage. The benefit of this is to recognize faults and misunderstanding in the initial stage of software design life cycle [3].

1.3 Scope of Formal Methods

Following are the three dimensions scope of the formal method use:

- The larger use of formal methods normally occurs in the early stage of life cycle.
- It is used in all or customized components of systems, where critical assessments is essential.
- The proofs and correctness are traditionally associated with formal methods to ensure the functional specification of a system and fulfill the requirements criteria. Therefore, formal methods are well applied to verify the most important parts of a system.

1.4 Formal Specification Language

A specification is a description of a product (either to be build or existing). Specifications are used in many different engineering disciplines including software engineering. In software engineering the products that are specified are software [7]. Associated with the notion of a specification, there is the notion of what it means for a product to satisfy (fulfill/meet/conform to/be compliant with) its specification.

Formal specification: The most common practice in software development is to use informal specifications, i.e. specifications written in natural language or using some diagrams or pseudo code. A complement to informal specification is formal specification. A specification is said to be formal if it is expressed in a formal notation or a formal specification language, i.e. a language that has a precise syntax (e.g. given by a BNF grammar) and for which every sentence in the language has a unique mathematical meaning. The underlying mathematical concepts are often simple, e.g. being based on mathematical logic and set theory.

What are formal specifications used for: Formal specifications are, just as informal specifications, used in the analysis and design phases of the software development cycle to record requirements and design decisions, respectively. They can be used as contracts or communication media between customer and developers, and between developers. Besides being used as a base for design and implementation, formal specifications can also be used as a base for generating test cases, for simulation and for formal analysis of the described products in order to predict their behavior before they are implemented.

Why using formal specifications: The following subsections will further characterize formal specifications as

- being abstract (and subject to refinement),
- being precise, and
- allowing for formal (mathematical) analysis and explain the major advantages of that.

1.4.1 Abstraction and refinement

Specifications are characterized by being abstract in the sense that they omit details that are not relevant for their purpose. For instance, architects use floor plans, as the one shown in Figure 2, as specifications of buildings. The floor plan is an abstraction of a building – for instance it omits details about the building materials. Similarly, in software engineering, specifications omit implementation details about the software they describe. How much is omitted depends on where in the software development cycle the specification is used: generally more is omitted in the earlier phases.

In the requirement analysis phase of the software development cycle a (requirement) specification is created. This specification should give a description of the requirements to the software system to be implemented. A good requirements specification describes what the software system should fulfill (in the form of some desired properties), and not how it should accomplish that. It should concentrate on essential details and defer complicated implementation details to later phases.

As a consequence of abstraction, a specification may have several possible implementations, i.e. there may be several products that satisfy (i.e. meet or conform to) the specification. For instance, in the architect example, a floor plan may be implemented by several buildings differing with respect to the choice of building materials, as also illustrated in Figure 2.

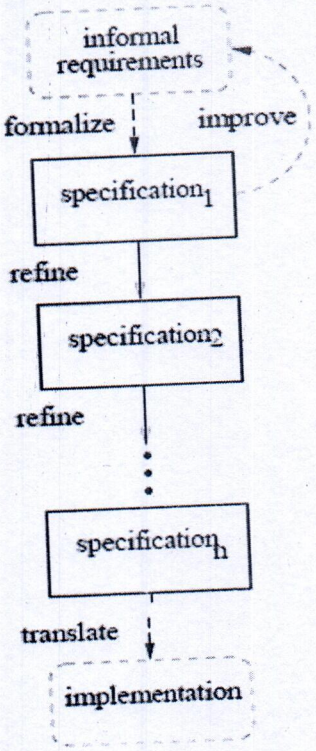

Fig 2: Stepwise refinement of specifications

1.4.2 Precision

Often specifications are expressed in natural language, or using some diagrams. However, such informal specifications have the disadvantage that they may be ambiguous, and/or (unintentional) incomplete. If a formal specification had been used instead, the specifier would have been enforced to make a decision on that and express it in the specification. Generally, the use of mathematics in formal specifications has the advantage that it enforces the specifier to think deeply about the problem, remembering all possible cases. Furthermore, it has the advantage that the resulting formal specifications are unambiguous as they have a unique mathematical meaning, and hence they are not open to different interpretations. While formal specifications have these advantages of being precise, informal specifications have the advantage of being more intuitive. Therefore, it is recommended to combine informal specification with formal specification, obtaining the best of each. In the requirement specification phase this is typically done by first capturing and formulating the requirements informally, and then formalizing the informal requirements. Experience also shows that this formalization process often leads to the discovery of ambiguities, incompleteness and inconsistencies in the informal requirements. After the discovery of such problems, they should be resolved so that the formal specification can be made and the informal description of requirements should be changed accordingly as illustrated by the dotted arrow named improve in Figure 2. As a consequence also the informal requirements become more precise Formal (mathematical) analysis In contrast to informal specifications, formal specifications can be mathematically analyzed as they are mathematically based. Hence, one can use mathematics to prove properties of a formal specification in orde1 to validate the specification and one can verify refinements of specifications [54].

1.5 Type of Formal Specification Languages

1.5.1 Sophistication-Based Formal Specification Languages

Formal techniques can be heavy or lightweight [8]. The light weight techniques [9] do not require advance mathematical background and theorem proving skills for usage of them. Formal methods techniques that include the feature of light weight are Z-notation and VDM tools. The heavy-duty approach of formal methods uses advance mathematics and theorem proving skills PVS and AC12 [15] are examples of heavy weight techniques.

1.5.2 Semantic-Based Formal Specification Languages

Formal methods based on semantic foundation are most commonly used for formal specification [5]. Model oriented and properties oriented are two main semantic based approaches of formal methods. VDM, CSP [10], Statcharts [16], RAISE [13], Petri nets [11] and B-Method [12] are common examples of model oriented formal methods.

On the other hand, property oriented formal methods enable minimal constraints on the system's behavior, which are necessary. The internal structure of system is not prescribed in detail.

1.5.3 Application-Based Formal Specification Languages

According to the system's nature, the different approaches are applied on the system for specification [5]. With respect to the applicability, the formal languages are different from one another. VDM, Z and Larch are used for sequential formalism whereas the Petri net and LOTOS [14] are used for concurrent modeling processes.

1.5.4 Graphical-Based Formal Specification Languages

Petri-nets and SDL [17] are two well-known graphical approaches, which are used for formal specification of systems. Data flow diagram (DFD) [18] and Unified modeling language (UML) are two semi formal methods techniques which are graphical based. Finite state machine (FMS) is considered as formal specification technique.

1.6 Benefits of Formal Methods

Formal methods have number of benefits; some of them are given below:

- Formal methods eliminate errors and bugs, which are found inevitable in informal specifications. By eliminating these ambiguities, readers and writers have consistency in understanding the specifications, and requirements are implemented correctly.
- Formal methods provide formal proofs of the given formal specification that removes the ambiguities, deadlock and inconsistency from the requirement analysis.
- The formal proofs and formal specification both provide repeatable and systematic approach for analysis.
- Formal specification and proofs both are applied in any stage of software development life cycle.
- The flow and syntax in formal proofs and specification can also be checked by computer based tools.

1.7 Different levels of formal methods:

Using formal methods does not necessarily mean that one should make everything formal. The use of formal methods can be more or less elaborate. The use can be classified at three levels according to how formal the specification and verification activities are:

1. Formal specification
2. Formal specification and semi-formal verification
3. Formal specification and formal verification

The choice of which level to use in the development of an application should be decided based on how critical the application is and the available resources (time, money, people having the right skills etc.) One may also choose to use different levels for different components to be developed. For instance one could decide to make a formal specification of the whole system, but only to use formal verification for one critical function.

2. Z-Notation

Z has been developed at Oxford University since the late 1970's by members of the Programming Research Group (PRG) within the Computing Laboratory. It is a typed language based on set theory and first order predicate logic.
The Z notation is a model oriented approach based on first order predicate logic [20] and Zermelo-Fraenkel (ZF) set theory [19] used for specifying behavior of abstract data types and sequential programs. The notation was originated and inspired by Jean-Raymond Abrial while visiting the Oxford University Computing Laboratory, and was subsequently further developed by Hayes, Morgan, Sorensen, Spivey, Sufrin and others [22]. It is undergoing international standardization under ISO/IEC JTC1/SC22 [21]. Z is usually used for systems development because it describes state space of a system and operations which can be performed over it.

2.1 Syntax

Z notation is basically based on:

- **First Order Predicate Logic**
- **Set theory and Relation**

2.1.1 First-order logic

First-order logic is symbolized reasoning in which each sentence, or statement, is broken down into a subject and a predicate. The predicate modifies or defines the properties of the subject. In first-order logic, a predicate can only refer to a single subject. First-order logic is also known as first-order predicate calculus or first-order functional calculus [21]. There are two logical constants in predicate logic, namely *true* and *false*. The basic elements of First Order Predicates are:

- **Constants** **KingJohn, 2, NUS...**
- **Predicates** **Brother, >...**
- **Functions** **Sqrt, LeftLegOf...**
- **Variables** **x, y, a, b...**
- **Connectives** $\neg, \Rightarrow, \wedge, \vee, \Leftrightarrow$
- **Equality** $=$
- **Quantifiers** $\forall, \exists$

Connectives

The propositional connectives of negation, conjunction and disjunction are used.

Table1: Connectives

Notation	Name	Explanation
$\neg P$	negation	*true* iff P is *false*
$P \wedge Q$	conjunction	*true* iff P and Q are both *true*
$P \vee Q$	disjunction	*false* iff P and Q are both *false*

Quantifiers

Allows statements about entire collections of objects rather than having to enumerate the objects by name.

•Universal quantifier- ∀x

Asserts that a sentence is true for all values of variable x

∀x Loves(x, FOPC)

∀x Whale(x) ⇒Mammal(x)

∀x Grackles(x) ⇒ Black(x)

∀x (∀y Dog(y) ⇒ Loves(x,y)) ⇒ (∀z Cat(z) ⇒ Hates(x,z))

•Existential quantifier- ∃

Asserts that a sentence is true for at least one value of a variable x

∃x Loves(x, FOPC)

∃x (Cat(x) ∧ Color(x, Black) ∧ Owns (Mary, x))

∃x (∀y Dog(y) ⇒Loves(x, y)) ∧ (∀z Cat (z) ⇒Hates(x, z))

2.1.2 Set theory and Relation

Set

Set theory is a basis of modern mathematics, and notions of set theory are used in all formal descriptions. The notation used here is based on Zermelo-Fraenkel theory in which there are only sets. Members of sets can only be other sets. The word \element" may be used loosely when referring to set members treated as atomic, without regard to their set nature, and abstracting from their representation [23].

Basic set operations

Zermelo-Fraenkel set theory constructs its repertoire of set operations starting with the axiom of empty set, then showing how to build up sets using the axioms of pairing and of union, and trimming back with the axiom of subset or separation.

Table 2-Set extensions and unions in metalanguage

Notation	Name	Explanation
$\varnothing$	empty set	$x \in \varnothing$ always false
$\{\}$	empty set	$= \varnothing$
$\{x\}$	singleton set	$y \in \{x\}$ iff $y = x$
$S \cup T$	union	the set of x such that $x \in S \vee x \in T$
$\{x, y, ..., z\}$	set extension	$= \{x\} \cup \{y, ..., z\}$
$\{i : S \mid P(i)\}$	comprehension	subset of elements i of S such that $P(i)$, by axiom of separation
$S \cap T$	intersection	$\{x : S \mid x \in T\}$
$S \setminus T$	difference	$\{x : S \mid x \notin T\}$

Powerset

The axiom of powers asserts the existence of powersets. The set of all finite subsets is a subset of this. The notation for these two forms is shown in Table3

Table3: Powerset in metalanguage

Notation	Name	Expianation
$\mathbb{P}\, S$	set of all subsets	$T \in \mathbb{P}\, S$ iff $T \subseteq S$
$\mathbb{F}\, S$	set of all finite subsets	the smallest set containing the empty set and all singleton subsets of S and closed under the operation of forming the union of two sets

Numbers

Numbers are not primitive in Zermelo-Fraenkel set theory, but there are several well established ways of representing them. The choice of coding is not specified here. There are notations to measure the cardinality of a finite set, to define addition of natural numbers and to form the set of natural numbers between two stated natural numbers, as given in Table 4, where m and n stand for any expressions whose values are natural numbers:

Table 4: Operations on numbers in metalanguage

Notation	Explanation
$m + n$	sum of natural numbers m and n
$\# S$	cardinality of finite set S
$m \,..\, n$	set of natural numbers between m and n inclusive

Tuples and Cartesian products

In Z, tuples and Cartesian products may have two or more components. In this mathematical language, however, only pairs, which may be iterated, are used [24] [3].

Table 5-Tuples and Cartesian products in metalanguage

Notation	Explanation
(x, y)	pair
$x \mapsto y$	using "maplet", identical to (x, y)
first p	$first(x, y) = x$
second p	$second(x, y) = y$
$S \times T$	Cartesian product, set of ordered pairs whose first element is in S and whose second element is in T
$(x, y, ..., z)$	abbreviates $(x, (y, ..., z))$
$S \times T \times ... \times U$	abbreviates $S \times (T \times ... \times U)$

Relations

A relation is defined to be a set of Cartesian pairs. There are several operations involving relations, who are given equivalences in Table 5, where Q and R are any relations, and S any set. A proposition about relations is given in Table 6:

Table 6-Relations in metalanguage

Notation	Name	Definition
$id\ S$	identity function	$\lambda\ x : S \bullet x$
$dom\ R$	domain	$\{p : R \bullet first\ p\}$
$R^{\sim}$	relational inversion	$\{p : R \bullet second\ p \mapsto first\ p\}$
$S \lhd R$	domain restriction	$\{p : R \mid first\ p \in S\}$
$S \lhd\!\!\!\!- R$	domain subtraction	$\{p : R \mid first\ p \notin S\}$
$R(\!\mid S \mid\!)$	relational image	$\{p : R \mid first\ p \in S \bullet second\ p\}$
$Q \fatsemi R$	relational composition	$\{q : Q; r : R \mid second\ q = first\ r \bullet first\ q \mapsto second\ r\}$
$Q \oplus R$	relational overriding	$((dom\ R) \lhd\!\!\!\!- Q) \cup R$

Functions

A function is identified with a particular form of relation, where each domain element has only one corresponding range element. The phrase \partial function" means exactly the same as the word \function" without qualification. Table 7 shows the various forms of function that are identified. In the table S and T are any sets, and the resultant expressions are sets of functions of various sorts.

Table 7-Functions in metalanguage

Notation	Name	Definition
$S \pfun T$	partial functions	$\{f : \mathbb{P}(S \times T) \mid \forall p, q : f \mid first\ p = first\ q \bullet second\ p = second\ q\}$
$S \fun T$	total functions	$\{f : S \pfun T \mid dom\ f = S\}$
$S \rightarrowtail T$	bijections	$\{f : S \fun T \mid f^{\sim} \in T \fun S\}$
$S \fin T$	finite functions	$\{f : \mathbb{F}(S \times T) \mid f \in S \pfun T\}$

2.2 Z Schemas

A boxed notation called 'schemas' is used for structuring Z specifications. Schemas are primarily used to specify state spaces and operations for the mathematical modelling of systems. For example, here is a schema called *StateSpace*:

StateSpace

$x_1 : S_1; \ldots; x_n : S_n$

$Inv(x_1, \ldots, x_n)$

This schema specifies a *state space* in which $x1, \ldots, xn$ are the state variables and $S1, \ldots, Sn$ are expressions from which their types may be systematically derived. $Inv(x1, \ldots, xn)$ is the state *invariant*, relating the variables in some way for all possible allowed states of the system during its lifetime [25].

For example: The 'Birthday Book'- is a system for recording birthdays. It uses the following basic types (or given sets):

[*NAME*, *DATE*]

The *state space* of the Birthday Book is specified by:

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

$$known = \text{dom}\ birthday$$

The *state variables* are *known* (a number of people's names) and *birthday* (unique dates associated with each known person's name). The 'invariant' property of this schema is:

known = dom *birthday*

i.e., every known person has a birth date associated with them. Z makes use of identifier decorations to encode intended interpretations.

The declaration ΔBirthdayBook describes the change of state: it introduces four variables known, birthday, known' and birthday'. The birthday and known are observations before the change of a state, whereas the birthday' and known' are observations of the state after the change. The variables name? and date? with question mark represent the input to the operation

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$birthday' = birthday \cup \{(name? \mapsto date?)\}$$

Now consider the specification of an operation used to find the birthday of an input name if it is known to the system.

FindBirthday

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$date! : DATE$$

$$name? \in known$$
$$date! = birthday(name?)$$

The declaration ΔBirthdayBook describes only the information. It does not change the state of the BirthdayBook after the operation: That is both the values of known and birthday remain same before and after the operation. The other variable date! ending with exclamation mark is used for an output. The schema FindBirthday takes a name as input and gives corresponding birthday as output. For this, the schema declared input variable name? of type NAME and output variable date! of type DATE. The predicate part checks the name in the set known and if it exists then it provides the appropriate birthday as output. There may be zero, one, or more people with the same birthdays on that day, to whom birthday cards should be sent

The schema Remind used to find which people have birthdays on today.

Remind

$\Xi BirthdayBook$
$today? : DATE$
$cards! : \mathbb{P}\, NAME$

$cards! = \{n : known \mid birthday(n) = today?\}$

The schema denoted by InitBirthdayBook specifies the initial state of the system. This initialized the birthday book with empty set known.

InitBirthdayBook

$BirthdayBook$

$known = \varnothing$

This schema describes a birthday book in which the set known is empty, consequently, the function birthday is empty too.
The above specification describes the mathematical framework for both the state space and the operations which are carried out on it. The data which appears in the system are specified in terms of mathematical data types such as sets and functions.

2.3 Petri Nets

Petri nets [30] [31] are a graphical, formal method applicable to a large variety of systems in which concurrency, dynamic behavior, synchronous and asynchronous communication, and resource sharing have to be modeled. The basic Petri net is composed of four elements: places, transitions, arcs and tokens. Due to their representational power, the elements can express various roles. For instance, places normally represent conditions, status, states or operations. Transitions are generally used to represent starting or stopping of events, which occurs to change the status of places. Arcs are the connections between places and transitions, and tokens can represent number of elements or the current availability of resources [26].
The formal definition of a *Petri net structure* and of a *Petri net* [31] are given as follows:
Definition 7.1 A *Petri net structure* is a 4-tuple $N = (P; T; Pre; Post)$, where:
$P = (p1; p2;:::; pn)$ is a finite set of places;
$T = (t1; t2; :::; tm)$ is a finite set of transitions;
Pre: $(P \times T)$ -> N is an input function that defines directed arcs from places to transitions, and N is the set of nonnegative integers;
Post: $(P \times T)$ -> N is an output function that defines directed arcs from transitions to places;
Definition 7.2 A *Petri net* is a 2-tuple $PN = (N; M\,0)$, where:
N is a Petri net structure, and
$M\,0$: P -> N is the initial marking.
If *Pre* $(p; t) = 0$, then there are no directed arcs connecting p to t. In a similar manner, if *Post* $(p; t) = 0$, there are no directed arcs connecting t to p. If *Pre* $(p; t) = k$ (the same holds for *Post* $(p; t) = k$), then there exist k parallel arcs connecting place p to transition t (transition t to place p). Graphically, parallel arcs are represented by a simple arc with a number k as label and representing the presence of k multiple arcs (this is also called the *weight* k of an arc).

The dynamic behavior (execution) is described by changing markings (from M to M'), which is represented by the firing of an enabled transition. A transition t is enabled if each place P that has an input arc to t contains at least the number of tokens equal to the weight of the arcs connecting P to t ($M(p) \geq Pre(p; t)$ for any $p \in P$).
The flow of tokens represents state changing. The firing of an enabled transition t removes from each input place pi the number of tokens equal to the weight of the directed arc connecting pi to t, and deposits in each output place p o. the number of tokens equal to the weight of the directed arc connecting t to p o. Therefore, the firing of a transition yields a new marking $M'(p) = M(p) - Pre(p; t) + Post(p; t)$ for any $p \in P$. The matrix $C = Post - Pre$ is called the incident matrix of the corresponding Petri net.
A graphical representation of a Petri net and its execution is shown in Fig. 1. Initially, places PA with two tokens and PB with one token are enabling transition t1. After the firing of transition t1, the tokens in PA and PB are removed, and one token is deposited in place PC, as the weight of the arc from t1 to PC is one.

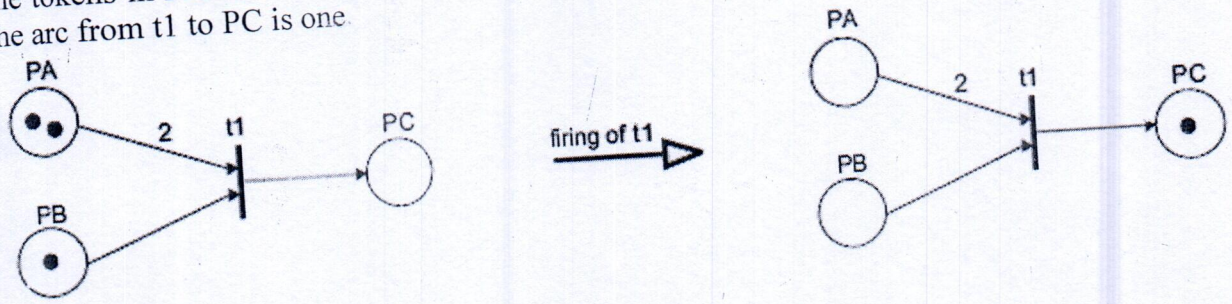

Figure 2.3: Execution of a Petri net

2.3.1 Properties of Petri Nets

The failure of software may have catastrophic consequences. For the correctness of a system, it is important that its design models are correct. Different concepts of correctness exist. For instance, a system is said to be correct when the specification and the actual implementation are equivalent [27]. Another definition for model correctness is that a model is correct when it exhibits a set of desirable properties [26]. These desirable properties allow the designers to identify the presence or absence of functional properties of the system under design. A complete list of Petri net properties can be found in [28] [31]. This subsection only briefly describes some of the most common ones.

- **Reachability:** In a dynamic system, it is fundamental to verify whether the system can reach a specific state as a result of a specific processing. In a Petri net, transitions firing will result in a sequence of markings that are reached with the change of token distribution. A marking Mf is said to be reachable from a marking Mi if there exists a sequence of firings that transforms Mi to Mf. The set of all possible markings reachable from Mi in a Petri net N is denoted by $R(N; Mi)$. The reachability problem for Petri nets is the problem of finding if $Mf \in R(N; Mi)$ for a given initial marking Mi.
- **Reversibility:** A Petri net is reversible if for every reachable marking $M \in R(N; M)$, the initial marking $M0$ is reachable from M. Thus, in a reversible Petri net, it is always possible to return to the initial marking or state.

- **Boundedness:** Given k a nonnegative integer, a Petri net is k-bounded if the number of tokens in each place does not exceed k for any marking reachable from $M0$, i.e., $M(p) \leq k$ for every place p and every marking $M \in R(N;M0)$.
- **Safeness:** A Petri net is safe if it is 1-bounded.
- **Liveness:** A Petri net is live if for every marking M that can be reached from $M0$, it is possible to fire any transition of the net by progressing through some further firing sequence. This means that a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen. In a Petri net model of a system, a deadlock is a marking M for which no transition is enabled. The fact that a Petri net is live ensures that for any reachable marking, at least one transition can be fired.

An important feature of Petri nets is the possibility of verifying the correctness of the designed model, i.e., to prove the presence or absence of properties. There are many methods to verify Petri nets, and these can be classified in different manners. A proposed classification of analysis methods given in [26] is as follows:

- Graph-theory,
- Linear Algebra,
- State-based methods,
- On-the-fly verification,
- Partial order methods and
- Logic

3. Scope of Work

When designing real-time systems for critical infrastructures, such as transportation and energy networks, system complexity is increased due to the large number of elements, strict real-time constraints, and reliability factors, as these systems involve human life. Therefore, it is necessary to use methods that are capable of addressing complexity and providing highly reliable solutions. A number of approaches have been used to model distributed real-time systems in general, and traffic control systems in particular. Finite State Machines (FSMs) have been used for modeling distributed real-time systems [29], but have the shortcoming of state explosion [32]. Statecharts [33] extends state-machines by endowing them with orthogonality, depth, and synchronization. An approach to model urban traffic signal control using Statecharts was proposed in [34]. Artificial Intelligence techniques such as Agents [35] [36] and Fuzzy Logic [37] have also been applied to model traffic signals control. Discrete Event System Specification (DEVS) [38] was used to generate optimal signal times based on the temporal sequence of traffic signal control variables and traffic information collected for each intersection [39].
Petri nets and Z/EVES are used to formally analyze safety critical properties of road traffic management system this in thesis .The choice is based on their modeling expressivity, their facilities for validation and verification activities, the availability of computer tools, and the range of available extensions [26] [27]. Petri nets are suitable to model distributed real-time systems because they offer representation of conflict situations, shared resources, synchronous and asynchronous communications, and precedence constraints whereas Z-notation is suitable for sequential behavior of system.